* **Kernel space** is the privileged execution space where the operating system core (kernel) runs. It has full access to hardware and memory.
* **User space:** This is where user applications (like your browser, media player, etc.) run. User space has **restricted access** to resources to avoid system crashes.

**copy\_to\_user()** → Used to **copy data from kernel space to user space**.

**copy\_from\_user()** → Used to **copy data from user space to kernel space**.

unsigned long copy\_to\_user(void \_\_user \*to, const void \*from, unsigned long n);

unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n);

| **Function** | **Parameter** | **Description** |
| --- | --- | --- |
| **copy\_to\_user()** | to | Pointer to user space buffer (\_\_user indicates user space) |
|  | from | Pointer to kernel space buffer |
|  | n | Number of bytes to copy |
| **copy\_from\_user()** | to | Pointer to kernel space buffer |
|  | from | Pointer to user space buffer (\_\_user indicates user space) |
|  | n | Number of bytes to copy |

**Return Value**

* 0 → if **all bytes were successfully copied**.
* **non-zero value** → Number of bytes that **could not be copied** due to:
  + Invalid memory access.
  + Partial memory access.
  + Kernel page fault.

**Reason #1: Page faults**

* If the user space memory page is swapped out or not allocated, dereferencing it will cause a **page fault**. This can lead to kernel crashes.

**Reason #2: Security**

* If a malicious user provides a pointer to **kernel space memory** (by mistake or on purpose), the kernel could accidentally overwrite critical data.

The first job of copy\_from\_user() is to **validate** the user space pointer.

unsigned long copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n)

{

if (access\_ok(from, n)) {

return \_\_copy\_from\_user(to, from, n);

}

return n;

}

The access\_ok() function is a **lightweight check** to ensure the user space pointer is valid. It ensures the user space memory **does not exceed user space limits**. Prevents kernel space from being accessed by mistake.

int access\_ok(const void \_\_user \*p, unsigned long size)

{

if ((unsigned long)p + size > TASK\_SIZE)

return 0;

return 1;

}

**TASK\_SIZE:** This is a constant defining the upper boundary of user space memory.On **32-bit systems**, it's typically 0xC0000000 (3GB). On **64-bit systems**, it's much larger.

unsigned long \_\_copy\_from\_user(void \*to, const void \_\_user \*from, unsigned long n)

{

return raw\_copy\_from\_user(to, from, n);

}

It uses **assembly instructions** like:

* + movsb (Move String Byte)
  + rep movsb (Repeat Move String Byte)
  + Kernel-mode DMA transfer (in some architectures).

If the user space page is **not present in memory**, a **page fault** occurs.

* + The kernel will: **Pause the copy. Load the page from disk to RAM. Resume the copy.**

This is handled via: **Copy-On-Write (COW)** mechanism. Kernel Page Fault handlers.

**Error Handling**

If any of the following happens:

* **Invalid memory access.**
* **Partial page present.**
* **Permission issue.**

The function will **abort** and return the **remaining number of bytes** not copied.

**How does copy\_to\_user() work internally?**

The flow is **exactly the same** except:

* The kernel now writes **data to user space**.
* The page fault handler writes data only to **writable user pages**.
* If the page is **read-only**, a fault will occur.

**Why Do Page Faults Occur in copy\_from\_user()?** Consider this user space application:

int fd = open("/dev/mydevice", O\_RDWR);

char buf[1024];

read(fd, buf, sizeof(buf));

When the user calls **read()**, the data must be **copied from kernel space to user space**. However, the **user space buffer (buf)** may point to:

* 1. **Unallocated memory** → Page fault.
  2. **Swapped-out memory** (moved to disk) → Page fault.
  3. **Kernel memory** (malicious or accidental) → Access violation.

**Call Sequence**

copy\_from\_user()

├── access\_ok()

├── \_\_copy\_from\_user()

│ ├── raw\_copy\_from\_user()

│ │ ├── Page fault may occur here

The CPU triggers a **page fault exception** when it detects:

* Invalid page access.
* Swapped-out page.

The CPU traps the fault and calls the kernel's **page fault handler**:

Page Fault

├── handle\_mm\_fault()

│ ├── do\_page\_fault()

│ │ ├── handle\_pte\_fault()

│ │ ├── bring page from disk (if swapped out)

│ │ ├── create mapping in MMU

│ │ ├── retry copy operation

**Page Fault Resolution (Major vs Minor)**

**Scenario 1: Page not allocated (Major Fault):** The kernel will **allocate a new page**. Map it in the **Page Table Entries (PTE)**. Resume the copy\_from\_user().

**Scenario 2: Page swapped to disk (Minor Fault):** Kernel will load the page from **swap space (disk)**. Restore it to **RAM**. Resume the copy\_from\_user().

**Scenario 3: Invalid Access:** If the user buffer is: **Kernel space**. **Completely invalid.**

* The kernel **returns an error** from copy\_from\_user().

**What Happens if Page Fault Occurs Too Frequently?**

* The kernel may:
  + **Kill the process** (if invalid pointer).
  + **Throttle the process** (if high page faults).
  + **Optimize memory usage**.

I'll now explain how **Direct Memory Access (DMA)** works when transferring data between **user space** and **kernel space** — and why it's vastly different from copy\_from\_user().

**DMA** allows devices (like NIC, GPU, Disk, etc.) to directly read/write memory **without CPU intervention**. It is **much faster** than copy\_from\_user() or memcpy().

**Why Is DMA With User Space Complicated?** In normal DMA: **Device → Kernel Memory → User Space**. But if you **directly map user space memory** to DMA, it eliminates an extra copy.

**Why Can't Devices Directly Access User Space Memory?**

* **Problem #1:** User pages may be swapped to disk.
* **Problem #2:** User memory is scattered (non-contiguous).
* **Problem #3:** Page Table Entries (PTE) are not available to devices.

**Kernel Mapping User Space for DMA:** The kernel must **pin user space memory** to:

* **Prevent it from being swapped.**
* **Translate virtual address → physical address.**

The kernel does this using:

int get\_user\_pages(struct task\_struct \*task, struct mm\_struct \*mm, unsigned long start, int nr\_pages, int write, int force, struct page \*\*pages, struct vm\_area\_struct \*\*vmas);

**Pin User Pages:** Suppose a user provides:

char \*user\_buf = malloc(4096);

write(fd, user\_buf, 4096);

The kernel must:

* **Pin pages** so they are never swapped out.
* Map them for DMA.

**DMA Map:** The kernel now uses:

dma\_map\_single(dev, page\_address(pages[i]), PAGE\_SIZE, DMA\_TO\_DEVICE);

This converts user space to **physical address**. Informs the device **where to transfer data**.

**DMA Transfer:** The device now does:

Device -> DMA -> Physical RAM

It bypasses the CPU entirely.

**DMA Unmap:** Once the transfer is done:

dma\_unmap\_single(dev, dma\_addr, PAGE\_SIZE, DMA\_TO\_DEVICE);

put\_page(pages[i]);

**Unpins the pages.** **Flushes cache if required.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Method** | **CPU Copy?** | **Cache Invalidation?** | **Swapping Risk?** | **Speed** |
| copy\_from\_user() | Yes | No | Yes | Slow |
| DMA with user space | No (device does it) | Yes (if required) | No (pinned memory) | Blazing Fast |

**DMA Is Used In:**

|  |  |
| --- | --- |
| **Device** | **Data Transfer** |
| Network Card | Huge data from NIC → User Space |
| GPU | Framebuffer directly to user space |
| SSD | Disk → User Space (Zero Copy) |

**Potential Problems With User Space DMA**

|  |  |
| --- | --- |
| **Problem** | **Why?** |
| Security Risk | Device may read user space secrets. |
| Page Faults | If memory is swapped, transfer fails. |
| Cache Incoherency | If cache isn't flushed, stale data may appear. |

**How Copy-On-Write (COW) Works During a Page Fault: Copy-On-Write (COW)** is a memory management technique where:

* **Multiple processes** can **share the same physical page** in memory.
* **Copying** the page happens **only if** one of the processes **modifies the page**.

**Why Is COW Needed?**

Consider this code:

pid\_t pid = fork();

if (pid == 0) {

// Child process

buffer[0] = 'A';

}

* **Parent and child** share the same memory (stack, heap, data).
* But **if one modifies the memory**, the kernel must:
  + **Copy the page**.
  + Give each process its **own private copy**.

**Fork() Without COW:** Before COW, the fork() would:

**Duplicate** the entire **address space** (all memory pages). Huge memory waste. Heavy CPU overhead.

**Fork() With COW:** Both **parent and child** share the **same physical memory page**. If any process **modifies** the page → Kernel performs **page fault** and copies the page.

**What Actually Happens During Fork()?**

**Parent Process Memory**

|  |  |  |
| --- | --- | --- |
| **Virtual Address** | **Physical Address** | **Data** |
| 0x1000 | 0xAABBCCDD | "Hello" |

**Child Process Memory (after Fork)**

|  |  |  |
| --- | --- | --- |
| **Virtual Address** | **Physical Address** | **Data** |
| 0x1000 | 0xAABBCCDD | "Hello" |

Both processes share the **same physical page** (0xAABBCCDD).Page is marked **Read-Only**.

**When Page Fault Occurs?** Suppose the child modifies:

buffer[0] = 'A';

* The kernel tries to write to the **shared page**.
* The page is **Read-Only** → **Page Fault** occurs.

**Handle Page Fault (COW Triggered)**

The kernel's **page fault handler** now executes:

Page Fault

├── do\_page\_fault()

│ ├── handle\_pte\_fault()

│ ├── cow\_page\_fault()

Inside cow\_page\_fault():

* **Allocate a new physical page**.
* **Copy the old data** to the new page.
* **Update the Page Table**.

**Result After COW**

|  |  |  |  |
| --- | --- | --- | --- |
| **Process** | **Virtual Address** | **Physical Address** | **Data** |
| Parent | 0x1000 | 0xAABBCCDD | "Hello" |
| **Child** | 0x1000 | **0xEEFF0011** | "Aello" |

Now **both have separate physical pages**. COW saves memory until the first **write** occurs.

**Why Is This So Powerful?**

|  |  |  |
| --- | --- | --- |
| **Scenario** | **Without COW** | **With COW** |
| Fork Overhead | High (full copy) | Almost zero |
| Memory Usage | High | Low |
| Page Faults | None | On write |

**What If No One Modifies the Page?** Both processes will **share the same page forever**. Huge memory savings.

**How Is This Related to copy\_from\_user()?** If copy\_from\_user() touches **COW pages**, it may **trigger page faults**. The kernel must first **resolve the COW fault** before copying.

**how cache flushing works** in the Linux kernel, especially during:

* **DMA (Direct Memory Access) operations.**
* **Cache coherency management.**
* **SMP (Symmetric Multiprocessing) scenarios.**

**Cache flushing** means forcing **dirty cache lines** (modified in cache but not in RAM) to:

* **Write back to RAM, remember not in hard disk.**
* Ensure memory consistency.

**Why Is Cache Flushing Needed in DMA?** Consider this:

* A **network card (NIC)** uses **DMA** to write data directly to memory.
* But the **CPU cache** may have **stale data**.
* If the CPU reads the old cache, it sees **invalid data**.

**Cache Types in Linux:** The two important caches in Linux:

1. **Write-back Cache (WB):**
   * **Most common.**
   * Modified data stays in cache → not immediately written to RAM.
2. **Write-through Cache (WT):**
   * Writes go **directly to RAM**.
   * Less performance, higher consistency.

**Step 2: When Does Cache Incoherency Occur?**

**Scenario #1: DMA From Device to Memory (NIC → RAM)**

Device (NIC) → DMA → RAM

* The device writes **directly to memory.**
* **CPU cache** still holds stale data.

**Scenario #2: CPU Modifies Data, Then Device Reads**

CPU → Cache → Memory

Device → DMA → Memory

* The **device** will read old memory if cache isn’t flushed.

**Step 3: Cache Flushing APIs in Kernel**

The kernel provides APIs like:

dma\_sync\_single\_for\_device()

dma\_sync\_single\_for\_cpu()

**What These Do:**

|  |  |  |
| --- | --- | --- |
| **API** | **What It Does** | **When To Use** |
| dma\_sync\_single\_for\_device() | Flush cache to memory | Before device reads memory |
| dma\_sync\_single\_for\_cpu() | Invalidate cache | After device writes memory |

**Assembly Behind Cache Flush:** On ARM:

dc civac, x0 // Data Cache Clean and Invalidate

**How Kernel Triggers Cache Flush in DMA**

dma\_map\_single();

dma\_sync\_single\_for\_cpu();

dma\_unmap\_single();

The internal path:

dma\_sync\_single\_for\_cpu()

├── arch\_sync\_dma\_for\_cpu()

│ ├── cache flush assembly

**Can Cache Flush Slow Down Performance? Yes.** Frequent cache flushes reduce performance. This is why **zero-copy networking** (like DPDK) tries to avoid it.

I'll now explain **how DPDK (Data Plane Development Kit)** achieves **zero-copy networking** by directly using **DMA with user space memory**.

**What Is Zero-Copy Networking?**

**Traditional networking**:

* **NIC → Kernel Buffer → User Space Buffer**.
* Involves **2 memory copies**:
  + **Device to Kernel**.
  + **Kernel to User Space**.

**Zero-copy networking**:

* **NIC → User Space Buffer (Directly)**.
* **No kernel involvement**.
* Achieves **insane throughput (up to 100 Gbps).**

**How Does DPDK Avoid Kernel Copy?** Instead of using:

copy\_from\_user()

DPDK:

* **Directly maps user space memory** to **DMA buffer**.
* Completely bypasses the kernel.

**Step 1: Memory Pinning**

DPDK first pins user memory using:

get\_user\_pages()

This prevents: **Page swapping and Page migration.**

**Step 2: DMA Mapping:** Then DPDK maps **user space memory** for DMA:

dma\_map\_single();

Which translates:

User Space Virtual Address → Physical Address

This allows **NIC** to directly write to user space.

**Step 3: Device Writes to User Space Buffer:** The device now:

* **Directly uses DMA**.
* Avoids kernel copy.

**Step 4: Cache Invalidation**

After DMA, the user space cache is stale. So DPDK does:

dma\_sync\_single\_for\_cpu();

Which forces the cache to:

* **Fetch from RAM (not cache).**
* Avoid stale data.

**Result: Insane Performance**

|  |  |  |
| --- | --- | --- |
| **Operation** | **Traditional Networking** | **DPDK Zero-Copy** |
| Kernel Copy | Yes | No |
| Interrupt Overhead | High | Zero (polling mode) |
| Throughput | ~10 Gbps | 100+ Gbps |

I'll now explain **how physical-to-virtual address translation** works during **Direct Memory Access (DMA)** in Linux. **What Is The Problem in DMA?**

**Normal Memory Access (Without DMA)**

* **User space virtual address → Kernel virtual address → Physical address.**
* Handled by the **MMU (Memory Management Unit)**.

**DMA (With Device Access)**

* Devices do not understand **virtual addresses**.
* They need a **physical address** to read/write data.
* The kernel must **map virtual memory to physical memory**.

**Step 1: Virtual Address Translation (Without DMA).** In a normal process:

* **User Space Address:** 0x7fff12340000
* **Physical Address:** 0x12345000

The MMU does:

Virtual Address → Page Table → Physical Address

The CPU handles it.

**Step 2: DMA Needs Physical Address**

Suppose the user does:

read(fd, user\_buffer, 4096);

The device does **DMA**, but:

* The device cannot understand 0x7fff12340000.
* It needs 0x12345000 (physical address).

**Step 3: Kernel Uses get\_user\_pages()**

The kernel calls:

get\_user\_pages();

This does:

* **Pin the page** in memory.
* Prevent swapping.
* Retrieve **physical page address**.

**Step 4: DMA Map**

The kernel now maps the **user page** using:

dma\_map\_single();

This:

* **Generates DMA physical address.**
* Makes the page accessible to devices.

**Step 5: Physical Address Used By Device**

Now the device can do:

Device → DMA → Physical Address (RAM)

**Step 6: Unmapping the DMA**

After the transfer:

dma\_unmap\_single();

put\_page();

This:

* **Unpins the page.**
* Flushes cache if necessary.